<https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html>

**Work Queues**



digraph { bgcolor=transparent; truecolor=true; rankdir=LR; node [style="filled"]; // P1 [label="P", fillcolor="#00ffff"]; Q1 [label="{||||}", fillcolor="red", shape="record"]; C1 [label=<C<font point-size="7">1</font>>, fillcolor="#33ccff"]; C2 [label=<C<font point-size="7">2</font>>, fillcolor="#33ccff"]; // P1 -> Q1 -> C1; Q1 -> C2; }

In the [first tutorial](https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html) we wrote programs to send and receive messages from a named queue. In this one we'll create a Work Queue that will be used to distribute time-consuming tasks among multiple workers.

The main idea behind Work Queues (aka: Task Queues) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a task as a message and send it to a queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

**Preparation**

In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the Thread.sleep() function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by Hello... will take three seconds.

We will slightly modify the Send.cs code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it NewTask.cs:

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

channel.BasicPublish(exchange: "",

routingKey: "task\_queue",

basicProperties: properties,

body: body);

Some help to get the message from the command line argument:

private static string GetMessage(string[] args)

{

return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");

}

Our old Receive.cs script also requires some changes: it needs to fake a second of work for every dot in the message body. It will handle messages delivered by RabbitMQ and perform the task, so let's call it Worker.cs:

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body;

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

};

channel.BasicConsume(queue: "task\_queue", noAck: true, consumer: consumer);

Our fake task to simulate execution time:

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Compile them as in tutorial one (with the client assembly in the working directory):

$ csc /r:"RabbitMQ.Client.dll" NewTask.cs

$ csc /r:"RabbitMQ.Client.dll" Worker.cs

**Round-robin dispatching**

One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

First, let's try to run two Worker.cs scripts at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the Worker.cs script. These consoles will be our two consumers - C1 and C2.

shell1$ Worker.exe

Worker

[\*] Waiting for messages. To exit press CTRL+C

shell2$ Worker.exe

Worker

[\*] Waiting for messages. To exit press CTRL+C

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

shell3$ NewTask.exe First message.

shell3$ NewTask.exe Second message..

shell3$ NewTask.exe Third message...

shell3$ NewTask.exe Fourth message....

shell3$ NewTask.exe Fifth message.....

Let's see what is delivered to our workers:

shell1$ Worker.exe

[\*] Waiting for messages. To exit press CTRL+C

[x] Received 'First message.'

[x] Received 'Third message...'

[x] Received 'Fifth message.....'

shell2$ Worker.exe

[\*] Waiting for messages. To exit press CTRL+C

[x] Received 'Second message..'

[x] Received 'Fourth message....'

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin.

**Message acknowledgment**

If one of the consumers starts a long task and dies with it only partly done, with our current code, once RabbitMQ delivers a message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it.

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

Message acknowledgments are turned on by default. In previous examples we explicitly turned them off by setting the noAck ("no manual acks") parameter to true. It's time to remove this flag and send a proper acknowledgment from the worker, once we're done with a task.

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body;

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);

};

channel.BasicConsume(queue: "task\_queue", noAck: false, consumer: consumer);

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered.

#### Forgotten acknowledgment

It's a common mistake to miss the BasicAck, and the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use rabbitmqctl to print the messages\_unacknowledged field:

$ sudo rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

Listing queues ...

hello 0 0

...done.

**Message durability**

We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as durable.

Although this command is correct by itself, it won't work in our present setup. That's because we've already defined a queue called hello which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example task\_queue:

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

This queueDeclare change needs to be applied to both the producer and consumer code.

At this point we're sure that the task\_queue queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by setting **IBasicProperties.SetPersistent** to true.

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

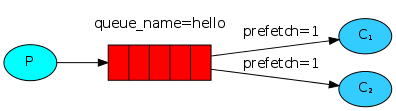
#### Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee then you can use [publisher confirms](https://www.rabbitmq.com/confirms.html).

**Fair dispatch**

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



digraph { bgcolor=transparent; truecolor=true; rankdir=LR; node [style="filled"]; // P1 [label="P", fillcolor="#00ffff"]; subgraph cluster\_Q1 { label="queue\_name=hello"; color=transparent; Q1 [label="{||||}", fillcolor="red", shape="record"]; }; C1 [label=<C<font point-size="7">1</font>>, fillcolor="#33ccff"]; C2 [label=<C<font point-size="7">2</font>>, fillcolor="#33ccff"]; // P1 -> Q1; Q1 -> C1 [label="prefetch=1"] ; Q1 -> C2 [label="prefetch=1"] ; }

In order to defeat that we can use the basicQos method with the prefetchCount = 1 setting. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

channel.BasicQos(0, 1, false);

#### Note about queue size

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

**Putting it all together**

Final code of our NewTask.cs class:

using System;

using RabbitMQ.Client;

using System.Text;

class NewTask

{

public static void Main(string[] args)

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

var message = GetMessage(args);

var body = Encoding.UTF8.GetBytes(message);

var properties = channel.CreateBasicProperties();

properties.Persistent = true;

channel.BasicPublish(exchange: "",

routingKey: "task\_queue",

basicProperties: properties,

body: body);

Console.WriteLine(" [x] Sent {0}", message);

}

Console.WriteLine(" Press [enter] to exit.");

Console.ReadLine();

}

private static string GetMessage(string[] args)

{

return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");

}

}

And our Worker.cs:

using System;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

using System.Text;

using System.Threading;

class Worker

{

public static void Main()

{

var factory = new ConnectionFactory() { HostName = "localhost" };

using(var connection = factory.CreateConnection())

using(var channel = connection.CreateModel())

{

channel.QueueDeclare(queue: "task\_queue",

durable: true,

exclusive: false,

autoDelete: false,

arguments: null);

channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);

Console.WriteLine(" [\*] Waiting for messages.");

var consumer = new EventingBasicConsumer(channel);

consumer.Received += (model, ea) =>

{

var body = ea.Body;

var message = Encoding.UTF8.GetString(body);

Console.WriteLine(" [x] Received {0}", message);

int dots = message.Split('.').Length - 1;

Thread.Sleep(dots \* 1000);

Console.WriteLine(" [x] Done");

channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);

};

channel.BasicConsume(queue: "task\_queue",

noAck: false,

consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");

Console.ReadLine();

}

}

}

Using message acknowledgments and BasicQos you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.

For more information on IModel methods and IBasicProperties, you can browse the [RabbitMQ .NET client API reference online](http://www.rabbitmq.com/releases/rabbitmq-dotnet-client/v3.2.2/rabbitmq-dotnet-client-3.2.2-client-htmldoc/html/index.html).

Now we can move on to [tutorial 3](https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html) and learn how to deliver the same message to many consumers.